資源制約スケジューリング問題 I

1 問題定義

資源集合 $\mathcal{R} = \mathcal{R}^{\mathrm{re}} \cup \mathcal{R}^{\mathrm{non}}$, 作業集合 $\mathcal{J} = \{1, 2, \cdots, J\}$, および処理モード集合 $\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_J$ が与えられ, さらに, 資源集合 $\mathcal{R}$ は再生可能 (renewable) 資源集合 $\mathcal{R}^{\mathrm{re}}$ と再生不可能 (nonrenewable) 資源集合 $\mathcal{R}^{\mathrm{non}}$ に分割されるとする. 再生可能資源 $r \in \mathcal{R}^{\mathrm{re}}$ は, 各単位時間 $[t-1, t)$ $(t = 1, 2, \cdots)$ に利用できる量 K^{re}_{rt} がそれまでのスケジュールに依らないのに対し, 再生不可能資源 $r \in \mathcal{R}^{\mathrm{non}}$ は, 各単位時間あたりではなく, スケジュール全体を通して利用できる量 K^{non}_{r} が決まっている. 例えば, 機械, 人, 作業場所などは再生可能資源, 原材料などは再生不可能資源と考えることができる。また, 処理モードとは作業の処理方法を表すものであり, 各処理モードに対して, 処理時間および処理に必要な資源量が決っている. 各作業 j はそれに対応する処理モード集合 $\mathcal{M}_j$ に属するいずれか一つの処理モードで処理され, 作業 j が処理モード m_j で処理されるとき, 処理時間を p_{m_j}, 作業の処理に必要な資源集合を $\mathcal{R}_{m_j} = \mathcal{R}^{\mathrm{re}}_{m_j} \cup \mathcal{R}^{\mathrm{non}}_{m_j}$ とする. また, 処理に必要な資源量は, 再生可能資源 $r \in \mathcal{R}^{\mathrm{re}}_{m_j}$ に対しては, 処理開始後各単位時間ごとに $k^{\mathrm{re}}_{m_j ru}$ $(u = 1, 2, \cdots, p_{m_j})$, 再生不可能資源 $r \in \mathcal{R}^{\mathrm{non}}_{m_j}$ に対しては, $k^{\mathrm{non}}_{m_j r}$ (作業の処理開始時刻には依存しない) であるとする.

スケジュールは, 各作業 j の処理モード $m_j \in \mathcal{M}_j$, および 開始時刻 s_j を用いて, $(\boldsymbol{m}, \boldsymbol{s}) = ((m_j \mid j \in \mathcal{J}), (s_j \mid j \in \mathcal{J}))$ で表される. また, 0-1 変数 x_{jm_j} を,

$$x_{jm_j} = \begin{cases} 1, \text{ 作業 } j \text{ が処理モード } m_j (\in \mathcal{M}_j) \text{ で処理される}, \\ 0, \ \text{ その他}, \end{cases}$$

と定義する. ここで, スケジュールに課せられる制約は以下のように分類される.

1. 資源制約

再生可能資源制約:

$$\sum_{j \in \mathcal{J}_{rt}} k^{\mathrm{re}}_{m_j r(t-s_j)} \le K^{\mathrm{re}}_{rt}, \qquad r \in \mathcal{R}^{\mathrm{re}}, t = 1, 2, \cdots, T. \tag{1}$$

ただし $\mathcal{J}_{rt}$ は, 資源 r を必要とし, 時間 $[t-1, t)$ に処理中である作業の集合 (すなわち, c_j を作業 j の完了時刻 $s_j + p_{m_j}$ としたとき, $\mathcal{J}_{rt} = \{j \mid r \in \mathcal{R}^{\mathrm{re}}_{m_j}, s_j + 1 \le t \le c_j\}$) であり, T は最大完了時刻 (makespan) $\max_j c_j$ の上界値である.

再生不可能資源制約:

$$\sum_{j \in \mathcal{J}} \sum_{r \in \mathcal{R}^{\mathrm{non}}_{m_j}} k^{\mathrm{non}}_{m_j r} x_{jm_j} \le K^{\mathrm{non}}_{r}, \quad r \in \mathcal{R}^{\mathrm{non}}. \tag{2}$$

本研究では, 再生不可能資源制約は, より一般的な不等式制約として記述される (後述の「3. その他の制約」参照).

2. 先行制約:

$$c_{j_1} \le s_{j_2}, \qquad j_1 \prec j_2. \tag{3}$$

ここで, $\prec$ は作業集合 $\mathcal{J}$ に対して予め与えられている先行関係である.

直前先行制約:
作業 j_1, j_2 $(j_1 \prec j_2)$ に対して,

$$\begin{array}{l} c_{j_1} \leq s_{j_2}, \\ s_{j_2} \leq s_{j'}, \quad j' \in \mathcal{J}, \text{ s.t. } c_{j_1} \leq s_{j'}, \end{array} \tag{4}$$

を直前先行制約 $j_1 \prec\prec j_2$ と定義し, さらに, 作業を再生可能資源 r 上に限定した制約

$$r \in \mathcal{R}^{\mathrm{re}}_{m_{j_1}}, r \in \mathcal{R}^{\mathrm{re}}_{m_{j_2}} \Rightarrow \begin{array}{l} c_{j_1} \leq s_{j_2}, \\ s_{j_2} \leq s_{j'}, \quad j' \in \mathcal{J}, \text{ s.t. } r \in R^{\mathrm{re}}_{m_{j'}}, c_{j_1} \leq s_{j'}, \end{array} \tag{5}$$

を資源 r 上における直前先行制約 $j_1 \prec\prec_r j_2$ と定義する.

3. その他の制約:

0-1 変数 x_{jm_j}, 開始時刻 s_j, 完了時刻 c_j, 処理時間 p_j $(= p_{m_j})$ に関する上記以外の制約. (再生不可能資源制約もこのタイプの制約として扱われる.)

現実のスケジューリング問題において, 評価基準は, 最大完了時刻や総納期遅れ時間など, 状況に応じて変化するため, 予め目的関数を定めておくことは好ましくない. また, より複雑な評価基準を用いることも多い. このため本定式化では, スケジュールの評価を制約を用いて行なう. すなわち, 上述の制約を, 必ず満たさなくてはならない制約 (絶対制約) $C^{\mathrm{hard}}_1, C^{\mathrm{hard}}_2, \cdots, C^{\mathrm{hard}}_{L_h}$ と, 満たすことが望ましいが必ずしも満たさなくてもよい制約 (考慮制約) $C^{\mathrm{soft}}_1, C^{\mathrm{soft}}_2, \cdots, C^{\mathrm{soft}}_{L_s}$ とに分類することができ, 絶対制約を満たす (実行可能である) 範囲で, 考慮制約の満足度を最大にするという意味でスケジュールを最適化する. これを実現するために, 各考慮制約 C^{soft}_l に対して, それが満たされる (満たされない) ならば 0 (正の値) をとるようなペナルティ関数 P_l, および制約 C^{soft}_l の重要度を示す重み w_l (> 0) を導入する. これらを用いて, 全体としての重みつきペナルティ関数 $P = \sum_{l=1}^{L_s} w_l P_l$ を目的関数とし, 最小化する. 各考慮制約に対するペナルティ関数は, 例えば, (左辺) $\leq$ (右辺) の形の不等式に対しては, $P_l = \max\{0, (\text{左辺}) - (\text{右辺})\}$ などとすればよい.

例: 最大完了時刻最小化

作業 sink の完了時刻 c_{sink} に対する等式制約

$$c_{\mathrm{sink}} = 0 \tag{6}$$

を考慮制約とし, c_{sink} をペナルティ関数とする. ここで作業 sink は, 全ての作業に後続する処理時間 0 の仮想的な作業である. 制約 (6) は, $(\forall j \in \mathcal{J}, p_j = 0$ でない限り) 満たされることはあり得ないが, 制約 (6) に関するペナルティを減らすことは, 最大完了時刻の減少を意味する.

2 アルゴリズム

本アルゴリズムは, 代表的なメタヒューリスティクスの一つであるタブー探索法に基づいている.

一般に, RCPSP の解は, 各作業 j の処理モードおよび開始時刻を表すベクトル $(\boldsymbol{m}, \boldsymbol{s})$ で表される. しかし効率的な探索を実現するため, 本研究では, 全作業の順列 (作業リスト) $\boldsymbol{\pi} = (\pi_1, \pi_2, \cdots, \pi_J)$ を用意し, 作業リストと処理モードベクトルの対 $(\boldsymbol{s}, \boldsymbol{\pi})$ の集合を探索空間とする. 解 $(\boldsymbol{s}, \boldsymbol{\pi})$ からは, $\boldsymbol{\pi}$ の順序に従って各作業 j の開始時刻 s_{π_i} を順に決定していくことによりスケジュールが生成される. このとき, 絶対制約のうち, 再生可能資源制約 (1), 先行制約 (3) および直前先行制約 (4)(5) のすべてを満たすようスケジュールは構成される. それ以外の絶対制約 C^{hard}_l は, 十分大きな重みを持つ考慮制約として扱い, それらを考慮に入れたペナルティ関数 P を目的関数として用いる. もちろん, 探索の途中, 絶対制約 C^{hard}_l を満たさない解を訪問することもあるが, 最終的に得られる最良解は C^{hard}_l を満たしていると期待できる.

解 $(\boldsymbol{m}, \boldsymbol{\pi})$ の近傍 $N(\boldsymbol{m}, \boldsymbol{\pi})$ は,

(i) ある作業 j の処理モード m_j を他の処理モード m'_j ($\in \mathcal{M}_j$) に変更する,

(ii) 順列 $\boldsymbol{\pi}$ において, ある作業 π_{i_2} を作業 π_{i_1} の直前に移す,

という局所的な変更によって得られる解の集合とする. ただし, 近傍内の解すべてを候補に探索を行うことは非効率的であるため, 本アルゴリズムでは, 現在の解 $(\boldsymbol{m}, \boldsymbol{\pi})$ が違反している制約 C_l に注目し, そのような制約のペナルティを減らす効果があると期待される局所的変更のみを試みる. そのために, クリティカルパスという概念を導入している.

また, タブー探索のパラメータである tabu tenure は, 探索中自動調節される.

3 例題

4 仕事 3 機械のジョブショップ問題を考える. 各仕事はそれぞれ 3 つの作業 1, 2, 3 から成り, この順序に処理しなくてはならない. 各作業を処理する機械, および処理日数は以下の通りである.

	作業 1	作業 2	作業 3
仕事 1	機械 1 / 7 日間	機械 2 / 10 日間	機械 3 / 4 日間
仕事 2	機械 3 / 9 日間	機械 1 / 5 日間	機械 2 / 11 日間
仕事 3	機械 1 / 3 日間	機械 3 / 9 日間	機械 2 / 12 日間
仕事 4	機械 2 / 6 日間	機械 3 / 13 日間	機械 1 / 9 日間

また, 各作業の初め 2 日間は作業員資源を必要とする操作であり, この操作は平日のみ, かつ 1 日あたり高々2 個しか行うことができないものとする. この他, 以下の条件・制約を考える.

- 機械 2 に限り, 特急処理が可能である. 特急処理を行うと処理日数は 4 日で済むが, コストがかかるため, 全体で 1 度しか行うことはできない.
- 機械 1 において, 仕事 1 を処理した後は仕事 2 を処理しなくてはならない.
- 目的は最大完了時間最小化である.

この問題は機械および作業員資源を再生可能資源として, 12 作業の RCPSP として記述できる. まず, 同一仕事内の作業間には先行制約が課される. 特急処理は, 通常処理とは別の処理モードとして表現され, 特急処理の回数制限は再生不可能資源制約 (3. その他の制約) として実現できる. また, 2 番目の制約は直前先行制約として扱うことができ, 目的関数は考慮制約 (6) として表現される.

4 プログラムの構成

プログラムの構成を図 1 に示す.

5 プログラムの使用法

本プログラムを用いて問題を解くためには, 以下のフォーマットに従い入力ファイルを作成する必要がある.

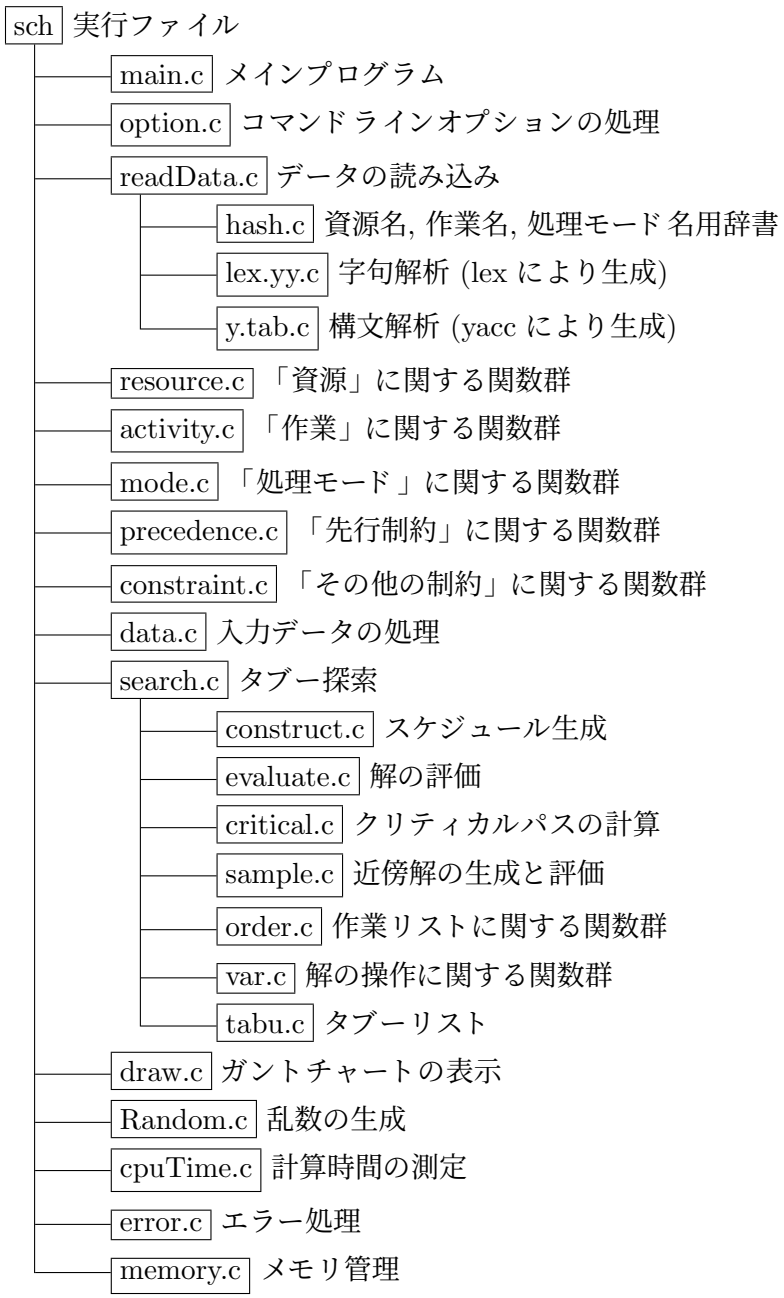

図 1: プログラムの構成

入力ファイルフォーマット

入力ファイルは

1. パラメータ記述部
2. 資源・作業・処理モード記述部
3. 制約記述部

で構成され, この順序に従って記述されなくてはならない. 以下, 各記述部の記述方法を説明する.

以下の説明中, *integer* は整数を表すものとし, 十分大きな整数を表す `inf`, 整数に対する二項演算 `+`, `-`, `*`, `Max(·,·)`, `Min(·,·)` および単項演算 `+`, `-`, `Abs(·)` の演算結果も整数として扱われる. ただし, (例えば `inf+inf` などに対する) オーバーフロー対策はしていない. *array* は整数列を表し, 各要素はカンマ (`,`) で区切られる. また, (*array*) `*` *integer* により, 部分列 *array* の *integer* (≥ 1) 回繰り返しを表

現できる (例えば, 0,1,2,2,2,1,2,2,2,1,0 は 0,1,((2)*3,1)*2, 0 と同じ). 繰り返し回数 *integer* に inf を指定することもできる. *string* は英字で始まり, 英数字・アンダーバー (_)・角括弧で括られた整数 ([*integer*]) からなる文字列を表す (例えば, *string* として job[1+1]_rsc[Abs(-2*2)] を与えると, これは job[2]_rsc[4] であると解釈される). ただし, inf, Max, Min, Abs, PARAMETER, PROBLEM, HORIZON, TARGET, ACTIVITY, RESOURCE, MODE, PRECEDENCE, CONSTRAINT, start_of, completion_of, time_of は予約語であり, *string* として使用することはできない. また, 入力ファイル中, # 以降その行末まではコメントとして無視される. コメント文以外においては, 改行は空白文字として扱われる.

1. パラメータ記述部

ここで設定されるパラメータとしては以下の 4 種類がある.

1.1. PARAMETER

PARAMETER *string* = *integer*

この宣言文により, 文字列 *string* を整数 *integer* として扱うことができる.

1.2. 問題名

PROBLEM *string*

問題例の名前を設定する. 省略された場合, ファイル名 (標準入力の場合 stdin) が問題例の名前となる.

1.3. スケジュール期間

HORIZON *integer*

スケジュール期間を (0, *integer*] に設定する. *integer* ≥ 0 でなくてはならない. 省略された場合, define.h で指定された値 HORIZON に設定される. 実行時, オプション -horizon で指定することもできる.

1.4. 目標値

TARGET *integer*

探索目標値を設定する. ペナルティが *integer* (≥ 0) 以下であるスケジュールが求まった時点で探索終了となる. 省略された場合, 目標値は 0 と見なされる. 実行時, オプション -target で指定することもできる.

2. 資源・作業・処理モード 記述部

RCPSP の基本要素である資源, 作業, および作業の処理モードを定義する.

2.1. 資源

RESOURCE *string* = { amount:*array1* weight:*array2* }

(再生可能) 資源 *string* を宣言し, 各単位時間における, 供給量および資源制約としての重みを設定する. ここで, 整数列 *array1*, *array2* の第 t 要素は, 時間 $(t-1, t]$ に対応する. 重みとして inf を指定すると絶対制約と見なされる.

2.2. 処理モード

MODE *string1* = { time:*integer* }
MODE *string1* = { resource:*string2 array* }

処理モード *string1* を宣言し, 処理時間, あるいは必要資源およびその必要量を設定する. ここで, *string2* は宣言済みの資源でなくてはならない. また, 複数の設定を 1 文で記述することもできる.

```
MODE string1 = {
  time:integer
  resource:string2 array1
  resource:string3 array2
  ...
}
```

2.3. 作業

ACTIVITY *string1* = { mode:*string2* }
ACTIVITY *string1* = { mode:{···} }

処理モード *string2*, あるいは{···}で定義される処理モードを作業 *string1* の処理モードとして追加する. *string2* は宣言済の処理モードでなくてはならず, {···}は, 処理モードの定義部 ("MODE *string* = " に続く{ }で囲まれた部分) でなくてはならない. また, 以下のように, 複数の処理モードを 1 文で設定することも可能である.

ACTIVITY *string1* = { mode:*string2* mode:{···} ··· }

注 意
全作業に先行 (後続) する仮想作業 source (sink) は自動的に定義されるため, ユーザがこれらを陽に宣言する必要はなく, source あるいは sink という名前の作業を宣言することもできない.

source および sink は, 処理時間 0 の処理モード dummy を唯一の処理モードとして持つ. 処理時間が 0 であるため, 実際には資源を消費することはないが, 便宜上, 全ての資源を必要とするものとする (直前先行制約のため).

なお, source, sink, dummy を, すでに宣言済みの作業, あるいは処理モードを表す *string* として用いることは可能である.

3. 制約記述部

再生可能資源制約は, RESOURCE による記述により与えられるため, ここでは, 先行制約およびその他の制約の記述を行う.

3.1. 先行制約

PRECEDENCE *string1* = { *string2* -> *string3* (time:*integer*) }
PRECEDENCE *string1* = { *string2* >> *string3* resource:*string4* }

先行制約 *string2* $\prec$ *string3* あるいは直前先行制約 *string2* $\prec\!\!\prec_{string4}$ *string3* を *string1* として与える. *string2*, *string3* は作業として, *string4* は資源として宣言済でなくてはならない. 先行制約の場合, time:*integer* を指定することにより, より一般的な時間制約 (*string2* の完了時刻) + *integer* $\leq$ (*string3* の開始時刻) を記述することができる. *integer* は負の値でもよく, 省略された場合 0 と見なされる.

本アルゴリズムでは, 先行制約, 直前先行制約は全て絶対制約として扱われる. 考慮制約として先行制約を記述する場合, 次に述べる「その他の制約」として記述する必要がある. 直前先行制約を考慮制約として扱うことはできない.

3.2. その他の制約

本アルゴリズムでは, 0-1 変数 x_{jm} を

$$x_{jm} = \begin{cases} 1, & \text{作業 } j \text{ が処理モード } m \text{ で処理される,} \\ 0, & \text{その他,} \end{cases}$$

とし, 変数 s_j, c_j, p_j をそれぞれ作業 j の開始時刻, 完了時刻, 処理時間として, これらに対する制約を記述することができる.

CONSTRAINT *string* = { weight:*integer* expression:*term1* *term2* $\cdots$ *sign* *integer* }

expression: により与えられる (不) 等式制約を制約 *string* として追加する. *string1*, *string2* をそれぞれ宣言済の作業, 処理モードとして, 各項 *term1*, *term2* $\cdots$ は

coeff [*string1* (*string2*)]
coeff [start_of *string1*]
coeff [start_of *string1* (*string2*)]
coeff [completion_of *string1*]
coeff [completion_of *string1* (*string2*)]
coeff [time_of *string1*]
coeff [time_of *string1* (*string2*)]

のいずれかでなくてはならない. *string1*, *string2* をそれぞれ作業 j, 処理モード m とすれば, これらは, それぞれ x_{jm}, s_j, $s_j x_{jm}$, c_j, $c_j x_{jm}$, p_j, $p_j x_{jm}$ の項を意味する. *coeff* は係数を表し, *integer* の他, 単に + や - でもよく, それぞれ +1, -1 であると見なされる. また係数は, 省略された場合 +1 であると解釈される. *sign* は, <=, =, >= のいずれかでなくてはならない.

ここで定義される制約は全て考慮制約となるため, 絶対制約として扱いたい場合, 十分大きな値を重みとして与えることで対応することになる (ただし, inf を重みとして与えることは, オーバーフローの原因となるので不可).

例として, 3 章の問題例を記述した入力ファイル (一部省略) を以下に示す.

```
HORIZON 100 # 省略可能（実行時に -horizon オプションで指定可能）

# 機械資源
RESOURCE machine[1] = { amount: (1)*inf  weight: (inf)*inf }
RESOURCE machine[2] = { amount: (1)*inf  weight: (inf)*inf }
RESOURCE machine[3] = { amount: (1)*inf  weight: (inf)*inf }
```

```
# 人的資源: 週末は使用不可
RESOURCE manpower = { amount: (2,2,2,2,2,0,0)*inf weight: (inf)*inf }

# machine[2] 用特急処理
MODE express = { time: 4
                 resource: machine[2] (1)*inf
                 resource: manpower   1, 1, 0, 0
}

# 作業の定義（activity[2]〜activity[4] は省略）
ACTIVITY activity[1][1] = {
        mode: { time: 7
                resource: machine[1] (1)*inf          # 機械資源は処理中常に必要
                resource: manpower    1, 1, (0)*inf   # 作業員資源は初めの２日のみ必要
        }
}
ACTIVITY activity[1][2] = {
        mode: { time: 10
                resource: machine[2] (1)*inf
                resource: manpower    1, 1, (0)*inf
        }
        mode: express # machine[2] 上の作業は特急処理可能
}
ACTIVITY activity[1][3] = {
        mode: { time: 4
                resource: machine[3] (1)*inf
                resource: manpower    1, 1, (0)*inf
        }
}
```

（中略）

```
# 先行制約
PRECEDENCE activity[1][1]_[1][2] = { activity[1][1] -> activity[1][2] }
PRECEDENCE activity[1][2]_[1][3] = { activity[1][2] -> activity[1][3] }

PRECEDENCE activity[2][1]_[2][2] = { activity[2][1] -> activity[2][2] }
PRECEDENCE activity[2][2]_[2][3] = { activity[2][2] -> activity[2][3] }

PRECEDENCE activity[3][1]_[3][2] = { activity[3][1] -> activity[3][2] }
PRECEDENCE activity[3][2]_[3][3] = { activity[3][2] -> activity[3][3] }

PRECEDENCE activity[4][1]_[4][2] = { activity[4][1] -> activity[4][2] }
PRECEDENCE activity[4][2]_[4][3] = { activity[4][2] -> activity[4][3] }

## 直前先行制約
#PRECEDENCE activity[1][1]_[2][2] = {
```

```
#        activity[1][1] >> activity[2][2] resource: machine[1]
#}

# 特急処理は高々1 回
CONSTRAINT num_express = {
     weight: 100
     expression:   [ activity[1][2] (express)] + [ activity[2][3] (express)]
                 + [ activity[3][3] (express)] + [ activity[4][1] (express)] <= 1
}
# 最大完了時刻最小化
CONSTRAINT makespan = { weight: 1 expression: [completion_of sink] <= 0 }
```

実行方法

make コマンドでコンパイル後,

```
% sch -help
```

でオプション一覧が表示される.

```
Usage: sch [-draw][-fixtenure][-horizon #][-initial filename][-input filename]
        [-interval #][-iteration #][-(no)oneviolation][-printdata][-printinfo]
        [-reduction #(%)][-repeat #][-report #][-save filename][-seed #]
        [-(no)sort][-target #][-tenure #][-time #.#][-verbose]
```

以下, これらのオプションについて説明する.

`-draw`
: ガントチャートを表示する. 詳細は省略する.

`-fixtenure`
: タブー期間 (tabu tenure) の自動調整機能を無効にする. デフォルト では有効.

`-horizon` n
: スケジュール期間を n に設定. デフォルトでは, 入力ファイル内で指定された値, 指定されていなければ, define.h 内で定義されている `HORIZON`.

`-initial` *file*
: 初期解を *file* により与える. *file* は, `-save` オプションにより作成されたファイルでなくてはならない.

`-input` *file*
: 入力ファイルを指定する. 省略された場合, 標準入力からの読み込みとなる.

`-interval` n
: 探索中, n 反復ごとに探索情報を出力する. 0 を指定すると探索情報は出力されない (ただし, 初期解, 最良値の更新などの情報は出力される). デフォルトは define.h 中の `INTERVAL`.

`-iteration` n
: 最大反復回数を設定する. デフォルトは define.h 中の `MAX_ITERATION`.

`-onecritical, -noonecritical`
: タブー探索の各反復においてクリティカルパスを辿る際, 任意に選んだ 1 本のみを辿るか (`-onecritical`), 全て辿るか (`-noonecritical`) を設定する. デフォルトは define.h 中の `ONLY_ONE_CRITICAL` が `TRUE` (`FALSE`) のとき前者 (後者).

`-printdata`
: 読み込んだ問題例のデータを出力する.

`-printinfo`
: 探索中の情報をやや詳しく表示する.

`-reduction` ρ
: 近傍探索の際, (クリティカルパスによって定義される) 近傍全体の ρ% のみを探索範囲とする. 100 未満の値を設定することで近傍探索を高速化することができるが, 一般に, 解の精度は低下する.

`-repeat` n
: (乱数系列の種を変えながら) 探索を n 回繰り返す. デフォルトでは 1 回のみ探索を行う.

`-report` n
: 探索中, 最良値が更新されたとき, その値が n 以下であれば, 出力を行う. デフォルトは define.h 中の `REPORT`.

`-save` *file*
: 探索終了時, 最良解を *file* に保存する. このファイルは拡張子 `.dat` を持つ. `-initial` オプションでこのファイルを指定することで, 初期解を指定することができる.

`-sort,-nosort`
: 考慮制約が複数存在し, かつ `-reduction` オプションによって近傍の縮小を行う場合のみ有効. 評価対象となる近傍解をランダムに選ぶか (`-nosort`), ペナルティが大きい考慮制約から得られるクリティカルパスによって定義される近傍解を優先的に選ぶか (`-sort`) を指定する. デフォルトは define.h 中の `SORT` が `FALSE` (`TRUE`) のとき前者 (後者).

`-seed` n
: 乱数系列の種を設定する. デフォルトは 1.

`-target` n
: 目標値を設定する. デフォルトでは, 入力ファイル中で指定された値, 指定されていなければ 0.

`-tenure` n
: タブー期間の初期値を n に設定する. デフォルトは define.h 中の `TENURE`.

`-time` n
: 最大計算時間を n 秒に設定する. デフォルトは define.h 中の `MAX_CPUTIME`.

`-verbose`
: 入力ファイルからのデータ読み込み状況を出力する.

3 章の問題例を解いたときの出力例を以下に示す. パラメータ等表示の後, 初期解がランダムに 30 (= define.h 内の `INITIAL_TRIAL`) 個生成される. それらの中でペナルティ最小の解を初期化としてタブー探索が開始される. 以下の例では, 22 回目の反復でペナルティ (makespan) 39 の解が得られ, それが最良スケジュールとしてプログラム終了時に出力されている. このスケジュールでは, 仕事 3 の作業 3 が特急処理されるのが分かる.

```
% ./sch -iteration 100 -interval 0 -input sample
# PROBLEM: stdin
# HORIZON: 100
# TARGET: 0
# max cpu time: 3.00[sec]
# max iteration: ---
# 1st seed: 1
# last seed: 1
# initial solution: randomly
# tabu tenure: adaptively controlled, initially set to 1
# 100% of candidates tested

# cpu time for reading data: 0.00[sec]
# cpu time for preprocessing: 0.00[sec]
# seed = 1
# generating 30 solutions...
  1: penalty = 153
penalty = 153 (iteration = 0, time = 0.00[sec])
  2: penalty = 260
```

（中略）

```
 30: penalty = 245
```

（初期解表示）

```
iteration = 0, time = 0.01, objective value = 45/45
penalty = 43 (iteration = 3, time = 0.01[sec])
penalty = 42 (iteration = 21, time = 0.01[sec])
penalty = 39 (iteration = 22, time = 0.01[sec])
********** best solution **********
source (dummy): 0 --> 0
activity[1][1] (noname1): 3 --> 10
activity[1][2] (noname2): 14 --> 24
activity[1][3] (noname3): 31 --> 35
activity[2][1] (noname4): 0 --> 9
activity[2][2] (noname5): 10 --> 15
activity[2][3] (noname6): 24 --> 35
activity[3][1] (noname7): 0 --> 3
activity[3][2] (noname8): 22 --> 31
activity[3][3] (express): 35 --> 39
activity[4][1] (noname10): 2 --> 8
activity[4][2] (noname11): 9 --> 22
activity[4][3] (noname12): 22 --> 31
sink (dummy): 39 --> 39
********** penalty **********
penalty = 39
# iteration = 22/100
```

```
# cpu time = 0.01/0.02[sec]
# tabu tenure = 3.62/7 (average/max)
```

参考文献

[1] K. Nonobe and T. Ibaraki, Formulation and tabu search algorithm for the resource constrained project scheduling problem, Essays and Surveys in Metaheuristics, edited by C.C. Ribeiro and P. Hansen, Kluwer Academic Publishers, pp.557–588, 2002.